Microsoft Corp.

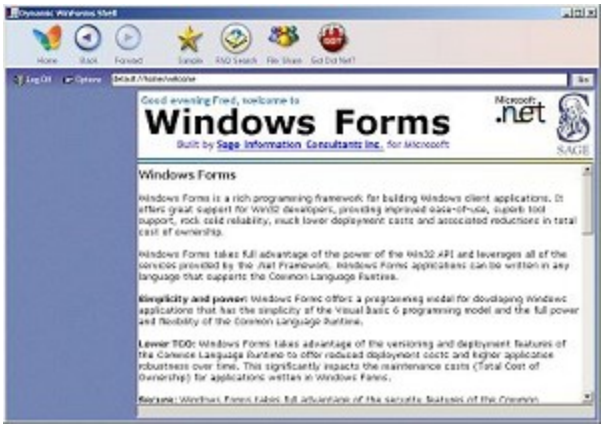
Enabling the Smart Client with the .NET Runtime

Mark Boulter
Program Manager
.NET Framework

May 13, 2001

Version 1 of the .NET Runtime removes key inhibitors to building applications that fully exploit the potential of the Windows Client PC – the Smart Client.

The Issue



My experience in talking to customers is that one big driver pushing people towards HTML and away from writing smart clients is the cost of deployment and the cost of maintaining a stable client machine over time. Installing applications onto client machines can be costly. Roll outs of new applications across several thousand desktops take too long. Additionally as more applications are installed the stability of the client machine degrades. Without a careful regression testing effort on a standard client machine, installing new applications can cause existing applications to stop working or crash.

This environment makes HTML-rendered applications running in the browser very attractive. These applications run in a standard environment, they are extremely easy to roll out, they are isolated from each other, they don't touch the global state of the client machine, they can be updated by simply updating the web server(s) and they can be personalized for a particular, person, group or role. It's not that HTML rendered applications are better apps, it's not that they are easier to develop (in many cases they are actually harder to develop), it's simply that they address the one of the biggest headaches our customers have today: deployment and application stability.

We understand why this is: "DLL Hell". In summary the "sharing by default" model introduced by DLLs and used by COM, the requirement to register components and so on has put us in a place where it's impossible not to share components and it's impossible to install an application without changing the global state of the machine. This leads to machine instability and more complex installation and deinstallation routines in order to effect global state changes in a controlled manner.

We have radically improved this situation with the improvements in Windows 2000, the Darwin installer and so on, but we have to get to the point where Smart client applications are as easy to install and as stable and predictable as HTML rendered applications.

Solving DLL Hell with the .NET Runtime

The .NET Runtime in conjunction with Fusion solves these problems for smart client applications.

Application stability = Application Isolation

The default model for applications and components is not sharing. By default applications are self contained and isolated, resolving components from the application bin directory rather than from a global or shared location. This means that applications are more stable over time - installing a new application does not affect any other applications on the machine, installing new applications does not affect shared components on the machine.

Applications are self describing and require no registry entries in order to run. This means that installing new applications does not affect the global state of the machine and that application install and uninstall can be significantly simpler. You can use XCOPY or FTP to simply copy the application to an appropriate directory and the application can then be run. You can simply delete the application directory and the application is completely removed from your machine (Note: If you want desktop shortcuts and program group entries you will still need a "real" installation process).

You can still share components that make sense to be shared between applications using the global assembly cache. However these components must be strong named - that is they must be uniquely identifiable. Multiple versions of shared components can be installed into the global assembly cache and by default applications explicitly bind to the strong name and version of a shared component that they were built against. Thus, even for shared components, applications are stable over time. (Note: You can change this binding policy via configuration files. This allows the application of hot fixes if required - the best of static and dynamic linking combined).

Installing a new app is no longer software Russian roulette.

Application deployment = Dynamic download of code over HTTP

The new component resolution model summarized above addresses application stability and predictability, and allows for simpler install and uninstall of applications, however it does not make applications as easy to install as HTML (navigate to a web page) or as dynamic in terms of updates to application components (install and new version on the web server and it automatically propagates to clients). The runtime addresses these issues by supporting dynamic download and execution of code across HTTP.

The Assembly class (in System.Reflection) enables download of assemblies over HTTP. This is relatively straightforward. Here is a simple example that shows downloading a form and displaying it:

```
Download the assembly from a web server over HTTP
Dim formAsm As [Assembly] = [Assembly].LoadFrom("HTTP://localhost/FirstForm.dll")

Get the Form from the assembly
Dim formtype As Type = formAsm.GetType("FirstForm.FirstForm")

Create an instance of the Form
Dim FormObj As Object
FormObj = Activator.CreateInstance(formtype)

'Cast it to a Form object to enable early binding and show the form
```

```
Dim Form1 As Form = CType(FormObj, Form)
Form1.Show()
```

The downloaded assembly is stored in isolation in the Assembly Download Cache. Different applications can download different versions of the assembly without causing conflicts. The assembly does not need to be registered on the client and so downloading and running the code in the assembly does not affect the global state of the client machine.

If the assembly is already in the download cache and the version on the web server has not changed then the assembly is loaded out of the cache. If the version on the web server has been updated, that new version is downloaded and executed. Thus updates are automatically propagated to the client but the network is not flooded with redundant downloads. This is exactly the same model as that used for HTML pages - in fact Fusion uses WinInet to download assemblies and so the technology used for downloads and versioning checking is the same in both cases.

Once the initial assembly has been downloaded from the web server, the runtime resolves any types in dependent assemblies by automatically downloading the assemblies from the web server and instantiating the types *only as needed*. I'll try to clarify what I mean with a simple example:

Let's say we have an application that consists of an exe installed on the client machine and 2 assemblies containing forms that are going to be dynamically downloaded over HTTP.

SecondForm.dll contains a form SecondForm. This assembly is stored in virtual directory on the web server.

FirstForm.dll contains a form FirstForm. This assembly is stored in virtual directory on the web server. FirstForm.DLL has a reference to SecondForm.DLL in its assembly manifest (a project reference in VS). FirstForm has a Button with a Click event handler that creates and displays an instance of SecondForm:

```
Dim f As New SecondForm.SecondForm()
f.Show()
```

MyApp.exe contains a form MainForm. MainForm has a Button with a Click event handler that calls *Assembly.LoadFrom* to download and display FirstForm (as above).

When the app is started, nothing is downloaded from the web server.

When the user clicks on the button in MainForm the Assembly.LoadFrom is executed and FirstForm.dll is downloaded and FirstForm is shown.

SecondForm.dll is not downloaded at this point because no code has been executed that causes the runtime to try and resolve the type.

Only when the user clicks on the button in FirstForm and the line of code `Dim f As New SecondForm.SecondForm()` executes does the runtime attempt to resolve the type. At this point the runtime examines the assembly manifest and discovers that SecondForm is in the assembly SecondForm.dll. The runtime then asks Fusion to resolve this assembly. Fusion does this by first looking in the same location as FirstForm.DLL - in this case the web server. It finds the assembly and downloads it to the client. SecondForm is then instantiated and shown.

The key points are:

- that only "entry point" assemblies need to be loaded using LoadFrom
- that dependent are only downloaded when they are referenced
- that dependent assemblies are automatically downloaded when needed

Combining these elements gives us a deployment model that brings smart client applications significantly closer to the HTML deployment model: no registration, dynamic application download, automatic propagation of updates, trickle feed of the application on to the client as its used - no 50MB, 1/2 hour install process just to use the first form.

Application personalization = Combining dynamic code download with web services

We are now in a position where customers will seriously reconsider using smart clients. So how can we enable building applications that are even more dynamic? One of the more interesting things that the HTML application model has enabled is personalization - the ability to tailor an application based on user preferences or user roles - for instance if I am a clerk in Accounting I get one set of "application tools", if I am a manager in Accounting I get another, if I am a Call Center Operative I should get another, if I am an Administrative Assistant I should get another and so on.

Once we have dynamic code download in place it turns out to be relatively easy to create more dynamic applications. For instance I can create a web service that returns information about the assemblies I have available on my web server:

```
<ArrayOfFormInfo>
  <FormInfo>
    <Name>Customers Form</Name>
    <Description>This allows you to maintain customer information including addresses and bill
payments</Description>
    <AssemblyName>CustomerForm.dll</AssemblyName>
    <TypeName>CustomerForm.CustomerUserControl</TypeName>
    <UrlLocation>HTTP://localhost/FormsListWebService/Forms/</UrlLocation>
    <UrlImage xsi:null="true"" />
  </FormInfo>
  <FormInfo>
    <Name>Orders Form</Name>
    <Description>This allows you to maintain order information</Description>
    <AssemblyName>OrdersForm.dll</AssemblyName>
    <TypeName>OrdersForm.OrderUserControl</TypeName>
    <UrlLocation>HTTP://localhost/FormsListWebService/Forms/</UrlLocation>
    <UrlImage xsi:null="true"" />
```

```
    </FormInfo>
</ArrayOfFormInfo>
```

I can then build a client application that uses this web service to build UI - for instance I can build an application that looks like Outlook, where the Outlook bar is populated using the Name, Description and Image properties. When the user clicks on an icon in the Outlook I can dynamically download the "Form" (in this case actually a User Control) using the rest of the information returned by the Web Service and display it as the "Current View".

I now have an application where all I install on the client is a simple shell app that can talk to a web service and populate its UI from that web service. The content of the application is completely driven by the information returned by the Web Service. A truly dynamic smart client application.

Once I have this application built it relatively easy to customize it on a per user basis. When the app starts up it prompts the user for a userid and password (or uses NTLM or Passport and so on). This information is passed to the web service and used to determine what assemblies the web service returns information about. I now have an application that is dynamically tailored for a specific user, group of users or user role.

Code Access Security = Making it safe to dynamically download code

Code access security is an important ingredient in these dynamic applications. They center on dynamically downloading code - it is very important that systems administrators feel safe to download and execute code on client machines. Code access security protects client machine resources and allows sys admins to enforce restrictions on downloaded code at run time.

Conclusion = The .NET Runtime and Framework enable smart clients

The combination of these features - application isolation, no registration, a strong versioning story, the capability to dynamically download code, automatic propagation of updates, trickle feed of the application on the client, the use web services to tailor the application - enables developers to build Smart client apps that have many of the important properties of HTML apps today using version 1 of the runtime.

As long as you have Windows clients and can afford to install the infrastructure on the client (.NET Runtime and frameworks and IE5.x) we have removed the key inhibitors to building smart client applications.

A Sample App that demonstrates these features

The purpose of the sample app is to:

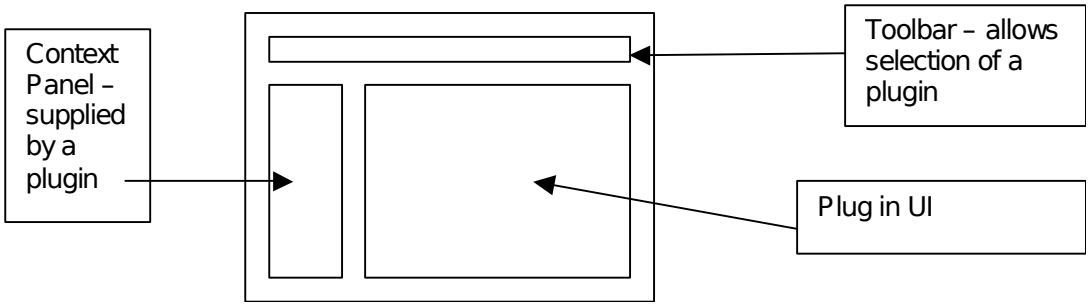
- Demonstrate the "no touch deployment" features of the.NET Runtime using Windows Forms - that is demonstrate the features described above.
- Demonstrate that it is possible to build "Cool UI" using Windows Forms - that Windows apps don't have to be boring grey forms.
- Demonstrate that the integration of Web Services into Windows Forms applications
- Demonstrate that you can use .NET to build peer to peer applications
- Demonstrate the security features of the .NET Runtime
- Demonstrate that Visual Studio.NET supports building applications of this type

The sample consists of a shell application - a "Windows Forms Explorer" modeled after MSN Explorer that hosts sub-applications or plugins - see picture at top of mail

Plugins are delivered to the Shell as assemblies (DLLs) loaded over HTTP. The plugins available for a given user are returned by a web service that is invoked when the user logs on. Each plugin adds a toolbar item to the Shell toolbar. When the users clicks a button in the toolbar the UI for that plugin is downloaded over HTTP.

Each plugin provides:

- A toolbar item to allow the plugin to be selected/started
- UI to be shown in the shell (typically built using Windows Forms)
- One or more "context" panels that are displayed whenever the plugin is active. Context panels give the user additional tools that can be used with the application



The first release of the sample includes the following plug-in:

- **Welcome page plug-in.** This plug-in acts as the "home" page for the shell application. It contains overview information about Windows Forms and pointers to more information.

- **.NET FAQ plug-in.** This plug-in displays an FAQ for .NET. It allows developers to browse the FAQ and search for particular items in the FAQ. Individual FAQs and the search facilities are delivered using a web service.
- **.NET Sample sharing peer-to-peer plug-in.** This plug-in offers a peer-to-peer sample sharing service that allows developers to share .NET samples they have written with their co-workers without having to upload them to a server or set up a network share on their machine.
- **GotDotNet plug-in.** This plug-in demonstrates the ability to integrate ASP.NET Applications into the Shell app (using the Web Browser control). The ASP.NET app is http://www.gotdotnet.com. It demonstrates the ability to add smart client "enhancements" to the ASP.NET app by providing additional options in the Context Panel.

This sample was built using Beta1 of Visual Studio and the .NET Framework. It uses Windows Forms, Web Services, typed DataSets (ADO.NET), the Net classes. The initial prototypes were built in VB.NET. This first version of the sample is built in C#. When we release the sample we will release VB and C# versions.